Object Oriented Metrics: Generation and Application

Mark Lattanzi*
Sallie Henry

Computer Science Department
Virginia Tech
Blacksburg, Virginia 24061
(703)231-6931
Fax: 703-231-6075

* **Contact Person:** Mark Lattanzi, lattanzi@cs.vt.edu, 703-231-6931

Abstract

Object oriented software has been cited by many [Cox86, Meye87, Coad90, Booc91] as a cure to some of the common problems occurring in software development. In particular, the object oriented paradigm, through inheritance, polymorphism, and encapsulation, allows for more reusable software components, easier software maintenance, and reduced software complexity. However, in order to make these claims, researchers will need to quantitatively measure various attributes of software (e.g. complexity, reusability, and maintainability). In order to efficiently and accurately use software metrics, it is necessary to automate the metric collection process. The metrics group at Virginia Tech has developed a tool (the Metrics Generator) that, through an interactive interface, allows software developers to generate various software metrics for their systems under development. This generator can generate metrics from Ada, C, Classic Ada, or C++ source. The most recent use of the tool was a study by Li and Henry [LiWe93] into the maintainability of object oriented code. The next use of the tool involves the object oriented paradigm and software reusability.

1. Introduction

Object oriented software has been cited by many [Cox86, Meye87, Coad90, Booc91] as a cure to some of the common problems occurring in software development. In particular, the object oriented paradigm, through inheritance, polymorphism, and encapsulation, allows for more reusable software components, easier software maintenance, and reduced software complexity. However, in order to make these claims, researchers need to quantitatively measure various attributes of software (e.g. complexity, reusability, and maintainability).

The standard way of measuring software is through software metrics. However, although metrics abound in the world of Software Engineering, most metrics were developed for use in the procedural paradigm. This paper briefly presents some of the more commonly accepted procedural software complexity metrics and then describes several new software metrics developed exclusively for use in the object oriented paradigm.

In order to efficiently and accurately use software metrics, it is necessary to automate the metric collection process. The metrics group at Virginia Tech has developed a tool (the Metrics Generator) that, through an interactive interface, allows software developers to generate various software metrics for their systems under development or under research. This generator generates metrics from Ada, C, Classic Ada, or C++ source. The generator has been used in several previous studies of software complexity [Henr90, Henr91, Lewi92, LiWe93]. The most recent study was performed by Li and Henry [LiWe93] on two commercial systems written in Classic Ada™.

The generator is currently being modified for a new study into the reusability of object oriented software. Hopefully, the metrics group can develop and validate some reusability metrics to allow for comparisons between fragments of object oriented software (classes) in terms of their reusability potential. The goal of this research is to provide software

Classic Ada is a trademark of Software Productivity Solutions, Inc.

developers with a better idea of which routines to include into ever-growing object-oriented software libraries, and which to leave out.

2. Traditional Metrics

2.1 Simple Metrics

At a very high level, the Metrics Generator reads in a set of source code files and outputs a set of metrics for each routine in the system. A routine is defined to be a high level language function, procedure, or method (in the case of object oriented languages). Classes can also be thought of as routines. Metrics can also be generated for the file, directory, or system level. The traditional metrics generated by the tool include three code based metrics: the number of lines of code [Cont86], some of Halstead's software science indicators (N, V, E) [Hals77], and McCabe's Cyclomatic Complexity [McCa76].

The number of lines of code has been used extensively in Software Engineering, so is included for comparison purposes. The definition for the Lines of Code metrics employed by this tool is based on the number of semi-colons in the source code [Cont86]. Halstead's software science indicators deal with the relationships between compiler recognizable tokens called operands and operands. The Metrics Generator implements N, V, and E. N is the total number of operands and operators in the system. V represents the volume of the program, the number of bits required to store the program in computer memory. $V = N * \log(n)$ where n equals the total number of unique operators and operands. The E indicator is a measure of the effort required to understand a piece of code. It depends on the program's volume V and its difficulty D. The effort (E) is the product of difficulty (D) and volume (V). D is calculated as follows: $D = n1 / 2 * N2 / n2$. where n1 equals the number of unique operands, n2 equals the number of unique operators, and N2 equals the total number of operands in the code. For a more detailed description of Halstead's metrics, see [Hals77].

The last code metric (created for the procedural paradigm) is McCabe's Cyclomatic Complexity. McCabe's metric is designed to measure the complexity and maintainability of a piece of code by counting the number of linearly independent paths through the code. This metric is based heavily on graph theory where sequential blocks of code represent nodes and decision statements represent the interconnections between the nodes. For a more complete description, see [McCa76].

2.2 Structure and Hybrid Metrics

The Metrics Generator also generates three other metrics: a structure metric and two hybrid metrics. The structure metric is McClure's Invocation Metric [McCl78]. The two hybrid metrics are Woodfield's Review Complexity [Wood80] and Henry and Kafura's Information Flow [Henr81]. Structure metrics attempt to measure the static interactions occurring in the code as well as the syntactic structure. McClure's metric attempts to quantify the fact the different predicates contribute varying amounts to the complexity of a module. McClure believes that in order to measure software complexity, the possible execution paths, the control structures, and the variables used to direct path selection must be included in the final metric.

Hybrid metrics are combinations of some code metrics with a structure metric. Woodfield's hybrid metric attempts to measure the effort required to understand a section of code in terms of time necessary to understand the code. Woodfield believes that a module must be reviewed several times before it can be completely understood. The number of times that a module needs to be reviewed depends on the number of flows into that module. The other hybrid metric, Henry and Kafura's Information Flow, attempts to quantify the interaction of the routines in the system by counting the number of entries (calls) to a procedure plus any references to global data. This value, called fan-in is multiplied to fan-out: the number of flows out of a routine plus the number of global variables that the routine updates. The final metric value is calculated by squaring the product of fan-in and fan-out. Intuitively, the larger the metric's value, the more complex the routine is likely to be.

3. Object Oriented Metrics

3.1 Overview

One of the projects of the Virginia Tech Metrics group involved modifying the Metrics Generator to output metrics for object oriented languages. The traditional metrics presented above continue to be generated by the Metrics Generator for object oriented languages, so the extent of their applicability in the object oriented paradigm can be determined. However, it is clear that a new suite of metrics must be developed and validated for object oriented languages.

The ten new object oriented metrics all deal with the class construct: a data structure that encapsulates the data and the functions (methods) that act on it into one logical unit. Classes can be constructed from other classes through instance variables or through inheritance. A discussion of classes with other classes as data items occurs later. Inheritance is the concept of deriving a class (the subclass) from another class (the super class) and then modifying certain parts of the super class's structure or adding new parts to the derived class's structure. A subclass is said to inherit all the properties of the superclass (the methods and the data structures). A superclass can have any number of subclasses (1 to many). Also, a subclass can have more subclasses derived from it (recursive).

Since metrics need to be gathered about each class in the system, the second output file generated during pass one of the generator contains the class hierarchy of the system.. Each class in the hierarchy contains information on the methods and data structures implemented by that class. From this data file, the ten new object oriented metrics are generated.

3.2 Ten Object Oriented Metrics

These ten new metrics can be similarly divided into two distinct groups: code metrics and structure metrics. The object oriented code metrics are similar to their procedural counterparts; they are generated by looking at the syntactic components of the code. The eight object oriented code metrics proposed by [Chid91] are:

1) Depth of Inheritance tree (DIT) This is a measure of the position of the class in the inheritance hierarchy. The lower a class resides in the hierarchy, the more super class properties the class can access. Intuitively, the larger (deeper) the depth of a class, the harder the class will be to maintain. The root of the class hierarchy has a depth of zero.

2) Number of Children (NOC): This is a count of the number of sub classes (children) directly derived from this class. The more children a class has, the more sub classes that are affected when the parent class has to be changed. Therefore, as NOC increases, so does the software maintenance effort (software complexity).

3) Response for a Class (RFC): This metric measures the cardinality of the response set of a class. The response set of a class is defined to be all the local methods of the class and all the methods called by local methods of the class. Intuitively, the more methods a class has or can call, the more complex the class will be.

4) Lack of Cohesion of Methods (LCOM): This metric measures the correlation between the methods and the local instance variables in a class. High cohesion indicates lower complexity. If LCOM is large, the measured class lacks cohesion and is more complex and harder to maintain. LCOM is measured by grouping the methods of the class into disjoint sets based on the local (class) variables that are modified by the methods. Methods modifying the same set of local variables are grouped into the same set.

5) Number of Local Methods (NOM): The metric can be viewed as an interface metric. NOM is a count of the number of local methods of a class (the class's interface). The idea is that if the interface to a class is large, it is more complex and harder to maintain.

6) Number of Semicolons (SIZE1): This metric is based on the lines of code metric. The more code that a class has, the more complex it tends to be. Although the lines of code metric has met with much resistance and criticism in the procedural paradigm, it is still widely used. We tend to view it as a baseline metric - a metric to perform better than.

7) Number of Methods plus Number of Attributes (SIZE2): This size metric is defined to be the number of local methods in a class plus the number of local data attributes. Again, the larger SIZE2 gets, the more complex the given class is likely to be. This metric was suggested in [LiWe93] by Dennis Kafura in an attempt to do better than just counting lines of code.

8) Weighted Method Complexity (WMC): WMC measures the static complexity of all the local methods of a given class and sums them. The static complexity measure can be any code metric. For our tool, we choose McCabe's Cyclomatic complexity as the base complexity metric. Clearly, classes with many methods or a few very complex methods are more complex than classes with simple methods or few methods.

The second type of object oriented metrics are the structure (coupling) metrics are closely related to the procedural structure metrics in that they are measuring the interactions (coupling) between the classes of the system. In more detail, these coupling metrics are:

9) Message Passing Coupling (MPC): This metric is used to measure the complexity of message passing among classes. Since the messages are defined by a class and used by objects of the class, the MPC metric gives an indication of how many messages are being passed among the objects of the classes. As the number of allowable messages increases, so does the coupling between this class and the rest of the system. Thus, the classes are more intertwined, and so, more complex and harder to modify.

10) Data Abstraction Coupling (DAC): This metric measures the number of instantiations of other classes within the given class. This type of coupling is not caused by inheritance or the object oriented paradigm. Generally speaking, any abstract data type with other abstract data types as members has data abstraction coupling. So, if a class has a local variable that is an instantiation (object) of another class, there is data abstraction coupling. Clearly, the higher the DAC, the more complex the data structures (classes) of the system.

Inheritance coupling measures the linkages created by the inheritance relations between classes. This type of coupling is already being measured by DIT and NOC, so it does not need to be measured again. However, if coupling is being measured as a separate attribute, the DIT and NOC code metrics could be combined into a new coupling metric.

4. The Metrics Generator

4.1 Overview

The metrics generator takes as input: **1)** a set of source code files (or directories), **2)** a BNF grammar for the source language (supplemented with semantic actions for each production rule), and **3)** an intrinsics file containing a list of all the functions intrinsic to the source language. Currently, there are fully working BNF grammars and intrinsics files for Pascal, C, Ada, and C++. The files exist for Classic Ada™, but are not yet completely integrated into the tool.

The generator outputs the three basic types of software metrics for procedural languages and the set of new object oriented metrics as well. The generator uses Lex and YACC in a three pass system to generate all of these metrics (See Figure 1). The first pass takes the three input files specified above and generates the code metrics and two intermediate files. One of these files contains a relational language equivalent to the source code [Henr88]. The other file is only generated for object oriented languages. It contains the class hierarchy information needed to generate the new object oriented metric suite. The generator currently runs under UNIX. Except for one module, the generator is written in C, and can therefore, be ported to other platforms easily.

4.2 The Metrics Generator: Pass One

Pass one of the generator uses Lex and YACC: two language parsing tools. First, the BNF grammar and the intrinsics file are used along with a library of (input) language independent routines to create a Metric Generator for the given input language (the source language). The generator can then read in and parse all the input source. After all the source is parsed, the generator outputs the three intermediate files.

The first file contains values of several metrics that can only be generated from the original source code. The code metrics include LOC, McCabe's Cyclomatic Complexity, and Halstead's Software Science Indicators (N, V, E). The second output file contains a relational language equivalent to the source code. For security reasons, source code is sometimes classified. This can make it difficult for researchers to gather metrics. Henry [Henr88] developed a relational language that is equivalent in complexity to the source language, but obscures the actual functionality of it. The third file produced by pass one is only produced for object oriented languages. It contains class hierarchy information necessary the produce our new suite of object oriented metrics.

4.3 The Metrics Generator: Pass Two

Pass two of the generator uses the relational language file produced in pass one to generate a set of specific relations which capture the procedural interactions contained within original source code. Again, Lex and YACC accomplish this translation. If the source language is object oriented, then the class hierarchy file generated from pass one is parsed and a detailed inheritance tree is constructed. From the inheritance tree, ten object oriented metrics are generated. Some of these metrics are based on the work done by Chidamber and Kemerer at MIT [Chid91]. Others were defined by Li and Henry [LiWe93] at Virginia Tech. All of these metrics measure software complexity at the class level. Briefly , they are:

1) Depth in the Inheritance tree (DIT)
2) Number of Children (NOC)
3) Message Passing Coupling (MPC)
4) Response for a Class (RFC)
5) Lack of Cohesion of Methods (LCOM)
6) Data Abstraction Coupling (DAC)
7) Weighted Method Complexity (WMC)
8) Number of Local Methods (NOM)
9) Number of Semicolons (SIZE1)
10) Number of Methods Plus Number of Attributes (SIZE2)

Detailed explanations of these metrics and how they measure software complexity occurred in part III of this paper. For a complete analysis, see [Chid91, LiWe93].

4.4 The Metrics Generator: Pass Three

Pass three serves two functions in the generator. First, it takes the specific relations generated in pass two and generates some structure metrics (Henry and Kafura's Information Flow, McClure's Invocation Metric). The second function of pass three uses the above structure metrics combined with the code metrics generated in pass one to produce several hybrid metrics (Henry and Kafura's Information Flow, Woodfield's Review Complexity).

The metrics generated by the tool can be output in many different formats from text files to bar graphs or charts. The text files can be formatted for easy input into common statistics packages (SAS™ , Quattro Pro™ , etc.).

4.5 The Metrics Generator: An Example

The following is a short example written in C++ and the output is from the final stage of the Metric Generator tool for the ten new object oriented metrics. The example consists of a short C++ program that contains four classes in one class hierarchy. The example is quite contrived and is included only to illustrate how to calculate the new object oriented metrics. The actual code does little real work. The class hierarchy (tree) of the example is shown in the first comment of the code.

```
/***********************************
Classes
                    Object
                      |
                      |
                  Collection
                    /   \
                   /     \
                 Set    Stack

#define error       int
#define boolean     unsigned char
#define OK          0
#define BAD         -1

//***********************************
// Object Class: Root of Class Hierarchy

class Object {
  public:
    virtual char *classname (void) { return "Object";
};
  };

//***********************************
// Collection Class: Inherits Object Class

class Collection : public Object {
  private:
    int getelemidx (int);
  protected:
    int size;
    int maxsize;
    int *data;
  public:
    Collection (void) {;}
    Collection (int);
    ~Collection (void) { free (data); };
    virtual error addelem (int);
    virtual error deleteelem (int);
    virtual boolean findelem (int);
    virtual int getelem (int);
        virtual char *classname (void) { return
"Collection"; };
    virtual boolean empty (void) { return !size; };
  };

Collection::Collection (int csize) {
  if ((data = (int *)malloc (csize)) == NULL) {
    printf("Out of Memory in Object Collection.\n");
    exit (BAD);
    };
  size = 0;
  maxsize = csize;
  };

error Collection::addelem (int i) {
  if (size == maxsize) return (-1);
  *(data+size) = i;
  size++;
  return OK;
  };

int Collection::getelem (int i) {
  if (i >= size) return (-1); else return *(data+i);
  };

int Collection::getelemidx (int i) {
  int j;
  for (j=0; j<size; j++)
    if (*(data+j) == i) return (j);
  return BAD;
  };

error Collection::deleteelem (int i) {
  int j, k;
  if (size == 0) return (-1);
  j = getelemidx (i);
   if (j == BAD) return (-1);   // element was not
found
  for (k=j; k<size; k++)
    *(data+k) = *(data+k+1);  // delete element i
  size--;
  return OK;
  };

boolean Collection::findelem (int i) {
  int j;
  for (j=0; j<size; j++)
    if (*(data+j) == i) return TRUE;
  return FALSE;
  };

//***********************************
//  Set Class: Inherits Object and Collection class

class Set: public Collection {
  public:
    Set (int i) : Collection (i) { ; };
    virtual error addelem (int);
    virtual char *classname (void) { return "Set"; };
    void print (void);
  };

error Set::addelem (int i) {
  if (this->findelem (i)) return OK;
  return (Collection::addelem (i));
  }
```

```
void Set::print (void) {
  int i;
  printf ("Set = {");
  for (i=0; i<size; i++)
    printf ("%d ", getelem (i));
  printf ("}\n");
  };

//***********************************
// Stack Class: Inherits Object and Collection
classes

class Stack: public Collection {
  public:
    Stack (int i) : Collection (i) { ; }
    error push (int i) { return (this->addelem(i)); }
    error pop (int *);
    error peek (int *);
     virtual char *classname (void) { return "Stack";
};
  };

error Stack::pop (int *i) {
  if (size == 0) return BAD;
  size--;
  *i = data[size];
  return OK;
  };

error Stack::peek (int *i) {
  if (size == 0) return BAD;
  *i = data[size-1];
   return OK;
   };
```

4.6 Output from the Metrics Analyzer for Each Class in the Example

Class	DIT	NOC	MPC	DAC	RFC	LCOM	WMC	NOM	SIZE1	SIZE2
Object	0	1	0	0	1	0	1	1	1	1
Collection	1	3	1	0	9	1	10	9	22	12
Set	2	0	2	0	7	4	2	4	7	4
Stack	2	0	1	0	6	5	2	5	8	5

5. The Object Oriented Paradigm and Software Maintainability

5.1 Description of Study

The most recent use of the Metrics Generator tool was performed by Wei Li in 1992. Li analyzed two commercial software systems written in Classic Ada™ (object-oriented Ada). The focus of the study was to bring together research in software metrics with research in the object-oriented paradigm. Specifically, Li investigated a suite of proposed object oriented metrics [Chid91], proposed several new object oriented metrics, and validated the metrics using maintenance data collected from two commercial systems. The ten metrics presented above were taken from Li's study.

Data was collected over a three year period on the maintenance effort required by the two commercial systems: QUES and UIMS. The maintenance effort was tracked by counting the number of lines changed per class. A change can be an insertion, a deletion, or an actual modification to a line of code.

Both of these systems are written in Classic Ada, a version of Ada supplemented with object oriented constructs (classes, inheritance hierarchies, etc.) Li analyzed both systems with the Metrics Generator. Based on these metrics, a prediction model was created. A statistical analysis of the prediction model was then performed relating the metrics to the actual maintenance effort. In this way, the suite of object oriented metrics was validated. The results of this study are presented below.

5.2 Results

After performing the study, Li came to the following conclusions.

1) Maintenance effort can be predicted in the object oriented paradigm.

The study of the two systems showed that a combination of the object oriented metrics predicted maintenance effort required. The maintenance data gathered correlated highly with the some of the ten metrics presented above. For a more detailed description of the actual multiple regression model, see [LiWe93].

2) Maintenance effort can be predicted from metrics gathered from design documents.

Although the ten object oriented metrics are code metrics, some of the metrics can be generated at design time due to extensive nature of some of the common object oriented design documents (e.g. the class hierarchy chart). Similar to result 1, when these ten metrics are used at design time, they correlate highly with the maintenance data.

3) There are observable patterns in the maintenance of object oriented systems.

Since there are patterns in the maintenance efforts, we can hope to predict what the effort will be given the existing code. These patterns were discovered using multiple linear regression models on the maintenance data and the ten metrics.

4) Object oriented metrics can be incorporated into an object oriented software development lifecycle.

Part of Li's study was to incorporate the use of these new metrics into the software development lifecycle. Plans for doing this were created and given to the company involved in the study, but no data is available on whether these new metrics were successfully integrated into the company's software development lifecycle.

6. The Object Oriented Paradigm and Software Reuse

Currently, the Virginia Tech Metrics Group is working in a relatively new area of Software Engineering: object oriented programming and software reusability. The object oriented paradigm has been shown to promote reuse in some case [Lewi92]. Object oriented libraries (databases) are currently available [GNU, Microsoft, Borland International, Smalltalk, etc.]. However, there are few, if any, metrics that attempt to measure the reuse potential of a piece of code (object oriented or otherwise). We are investigating what makes software reusable and how it can be measured. We are modifying our Metrics Generator to aid in this new research.

For our new research, two goals have been established. The first goal relates to the relative reusability between two pieces of code. Using the Metrics Generator, we should be able to order code fragments according to their reusability potential. The object oriented paradigm provides a good medium for testing since object oriented analysis and design promote reuse [Lewi92]. To achieve this goal, we are setting up an empirical study with college level seniors in Computer Science. The object oriented paradigm including object oriented analysis [Coad90, Booc91] and design [Booc91, CoxB86] will be taught to the subjects of the experiment. They will then be set to a task and their resulting products will be measured to aid in the development of our code metrics.

The second goal of this research is to be able to predict software complexity/reusability at design time in the object oriented paradigm. Clearly, to do this, some success in achieving goal one is required. Object oriented design is full of design time documents necessary for successful designs. These include class diagrams, object diagrams, state transition graphs, etc. [Booc91]. If these documents can be put into machine readable form, they can be analyzed to predict the complexity/reusability of the resulting code.

Again, using the same group of programmers, we can require certain design documents and attempt to correlate them with the final code in terms of complexity and reusability. Along with using student code, a set of object oriented libraries will be collected for the subjects' use. Subjective data on the relative reusability/complexity of the off-the-shelf reusable components provided in the libraries created by various software vendors can then be gathered from the experiment's participants. Using the opinions of the programmer's who had to use these object oriented libraries and using the Metrics Generator, our group should to be able to rank the components in the libraries according to their reusability.

After the empirical study, a suite of object-oriented metrics that will attempt to predict reusability from design documents and final code will be established. Next, this suite of metrics will need to be taken to a commercial software vendor for further validation and refinement in a "real world" setting.

7. Conclusions

Software Engineering has come a long way in the last two decades. A plethora of metrics exist for virtually every programming language. The job of the software engineer is to advance the state of the art in software development. We have constructed and used a tool to greatly aid the process of measuring software to help prove research hypotheses. The tool, the Metrics Generator, currently works for several programming languages, and the high level architecture is in place to easily expand the tool to incorporate other programming languages. A mechanism is also in place to circumvent the problem of measuring classified software [Henr88].

We have used the metrics generator on several previous software engineering studies with great success. The Metrics Generator greatly simplifies the process of gathering data about large software systems. We are currently working in the object oriented paradigm and the tool is being modified to handle the nuances of object oriented languages.

The Metrics Generator is a yet unfinished tool. It has been modified to handle new languages. A graphical user interface is being designed as a new front end for the tool. As new metrics are developed and validated, we will be incorporating them into the generator to further aid research and software development.

8. About the Authors

Mark Lattanzi

Mark Lattanzi received his B.S. in Electrical Engineering at Virginia Tech in 1987 with a concentration in digital design. He received his M.S. in Computer Science in 1989 from Virginia Tech under Dr. Clifford Shaffer. His primary fields of research include computer graphics and spatial data structures. Mr. Lattanzi's industry experience includes working for Hewlett Packard Company in Fort Collins, Colorado in the Graphics Technology Division. He developed low level graphics device drivers for HP personal computers and workstations. In 1991, he returned to Virginia Tech to pursue a Ph.D. in Computer Science under Dr. Sallie Henry. His current research interests focus on software metrics, the object oriented paradigm, and software reusability.

Sallie Henry

Sallie Henry received her B.S. from the University of Wisconsin-LaCrosse in Mathematics. She received her M.S. and Ph.D. in Computer Science from Iowa State University. After completing her Ph.D., she returned to the University of Wisconsin-LaCrosse as an Assistant Professor and later an Associate Professor of Computer Science.

Dr. Sallie Henry is currently an Associate Professor of the Computer Science Department at Virginia Tech. She has been working in the area of Software Engineering for the past eight years. Her primary interests are in Software Quality Metrics, Evaluation of Methodologies, Software Testing Methodologies, and Cost Modeling. Her most recent work has been in the validation of object oriented metrics to predict software maintainability, with particular focus on applying the metrics during the design phase of the software lifecycle. Dr. Henry's research has been supported by funding from the National Science Foundation, the Naval Surface Weapons Center, Digital Equipment Corporation, IBM, the Software Productivity Consortium, Virginia's Center for Innovative Technology, and Xerox Corporation. She is a member of IEEE and ACM.

9. References

[Booc91] Grady Booch, Object Oriented Design, The Benjamin/Cummings Publishing Company, RedWood City, CA, 1991

[Chid91] Chidamber, S.R. and Chris Kemerer, "Towards a Metrics Suite for Object Oriented Design", *OOPSLA 1991*, pp. 197-211.

[Coad90] Coad, Peter and E. Yourdan, Object Oriented Analysis, Prentice HAll, Englewood Cliffs, NJ, 1990.

[Cont86] Conte, S.D. and H.E. Dunsmore and V.Y. Shen, Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Co. Menlo PArk, CA, 1986.

[CoxB86] Cox, Brad, Object Oriented Programming: An Evolutionary Approach, Addison-Wesley, Reading, MA 1986.

[Hals77] M. Halstead, Elements of Software Science, Elsevier North Holland, New York, 1977.

[Henr90] Henry, Sallie and Calvin Selig, "Design Metrics Which Predict Source Code Quality", *IEEE Software*, March 1990.

[Henr81] Henry, Sallie and Dennis Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, September 1981, pp 510-518.

[Henr88] Henry, Sallie, "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers", *Journal of Systems and Software*, Vol. 8, No. 3, January 1988, pp. 3-11.

[Henr91] Henry, Sallie and Steve Wake, "Predicting Maintainability with Software Quality Metrics", *Journal of Software Maintenance*, Vol. 3, pp. 129-143, 1991.

[Lewi92] Lewis, John and Sallie Henry and Dennis Kafura, "On the Relationship Between the Object Oriented paradigm and Software Reuse", *Journal of Object Oriented Programming*, August 1992.

[LiWe93] Li, Wei and Sallie Henry, "Metrics for Object Oriented Systems, To appear in a special issue on the object oriented paradigm of the *Journal of Systems and Software*, November 1993.

[McCa76] McCabe, Thomas J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, December 1976, pp. 308-320.

[McCl78] McClure, Carma, "A Model for Program Complexity Analysis", *Proceedings 3rd International Conference on Software Engineering*, May 1978, pp. 149-157.

[Meye87] Meyer, Bertrand, "Reusability: The Case for Object Oriented Design", *IEEE Software*, March 1987, pp 50-64.

[Wood80] Woodfield, Scott, "Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors", Ph.D. Dissertation, Purdue University, Computer Science Department.